

Evaluating Tool Support for Co-Evolution of Modeling Languages, Tools and Models

Juha-Pekka Tolvanen

MetaCase

Jyväskylä, Finland

jpt@metacase.com

ORCID: 0000-0002-6409-5972

Steven Kelly

MetaCase

Jyväskylä, Finland

stevek@metacase.com

ORCID: 0000-0003-0931-157X

Abstract— We present a framework for evaluating language workbenches’ capabilities for co-evolution of graphical modeling languages, modeling tools and models. As with programming, language refinement, enhancement and other maintenance tasks typically account for more work than the initial development phase. Modeling languages have the added challenge of keeping tools and existing models in step with the evolving language. As domain-specific modeling languages and tools have started to be used widely, thanks to reports of significant productivity improvements, some language workbench users have indeed reported problems with co-evolution of tools and models. Our evaluation framework aims to cover changes across the whole language definition: the abstract syntax, concrete syntax and constraints. Change impact is assessed for knock-on effects within the language definition, the modeling tools, semantics via generators, and existing models. We demonstrate the viability of the framework by evaluating the MetaEdit+ tool, providing a detailed evaluation process for others to repeat with their tools. The results of the evaluation show that MetaEdit+ automatically updates and co-evolves models without error. In all cases the editors open and work with existing models; when automated co-evolution is impossible, the tool points to the items requiring human intervention. Industrial-scale experience with this approach, over language lifespans up to 25 years, is briefly assessed to corroborate its sustainability and evaluation.

Keywords—domain-specific modeling, domain-specific language, evolution, maintenance, metamodel evolution, model evolution

I. INTRODUCTION

Refinement, enhancement and other maintenance tasks normally account for more work than the initial development phase. This applies to domain-specific languages and models too, including their co-evolution. Compared to general purpose languages, domain-specific languages (DSL) and domain-specific modeling (DSM) languages evolve more frequently — following changes in the domain and in the development needs [1][2][3]. A recent DSL practitioner survey [4] found that 86% of respondents reported language evolution and recommended considering evolution as an intrinsic part of DSL creation.

An important characteristic of language evolution is that changes must be reflected in artifacts already made with the language: we want to preserve that work and move artifacts to the new language version. This enables sustainable development, both of the applications made by modeling, and of

the language itself. The economic and technical benefits of DSM co-evolution are clear, but there are also important benefits for other aspects of sustainability [5]. Poor co-evolution support can lead to language stagnation [2], harming communication and social sustainability as the distance between the stagnant language and its evolving domain increases. As the gulf widens, the 5–10 times productivity increase [1] offered by DSM falls, leading to increased resource usage in development. With ‘software engineer’ being one of the largest job categories these days, and IT equipment a significant consumer of energy, even environmental sustainability is at risk. Conversely, enabling co-evolution maintains the high sustainability benefits of DSM, from developer productivity to the ease of targeting new, lower-energy platforms with minimal effort through new generators.

The co-evolution of a domain-specific modeling language has an important characteristic due to its restricted use. If the language is made to address a narrow domain within a single company or its team, as reported in over a hundred cases [6], then it is likely that all language users are known, their specifications made with the language can be accessed, and data to assess the impact of language evolution can be inspected in all the language use contexts. Conversely, the number of users is significantly smaller than for general purpose languages, so the effort that can sensibly be spent per language change is smaller.

Research on co-evolution has focused on changes in certain parts of a language — such as in its metamodel or transformations — but not covered all aspects of a language together, as we aim to in this paper. Also, while most research on tools for DSLs and DSM, also called language workbenches [7], has focused on the initial steps of creating the language (e.g. [7][8][9]), we focus here on the refinement and maintenance of the languages and models made. We propose a framework that enables a holistic evaluation of a tool’s capabilities to support co-evolution. We apply the framework to show its viability by evaluating the MetaEdit+ tool [10]. For each evaluation task, detailed material is provided for others to repeat the evaluation process to validate it and to evaluate other modeling tools.

We start by describing previous research on language, model and tool co-evolution (Section II), and try applying an existing evaluation framework (III), leading us to suggest the set of aspects to include in our own framework (IV). Section V presents the procedure for applying our evaluation framework: an example language and model, along with a set of evolutionary steps to test all the aspects of co-evolution. In Section VI our

framework is then tested by following its procedure to evaluate the co-evolution support of MetaEdit+. We assess the evaluation framework based on the test experience (VII), and look at industrial experiences to corroborate the evaluation (VIII). In Section IX we conclude with proposed directions for future extension and verification.

II. RESEARCH ON CO-EVOLUTION WITH TOOLS

Research on co-evolution has focused on metamodels and models, with less research inspecting co-evolution of tool support, and mostly only experience reports mentioning both.

A. Research on Language and Model Co-evolution

The large body of work on co-evolution has focused on co-evolution of metamodels and models without considering evolution in other parts of the language definition, such as its constraints, notation or generators/transformations. A prevailing approach [11] has been to create transformations acting upon models (e.g. [12][13][14][15]) to enable their co-evolution with metamodel changes. Once defined, an appropriate transformation would be executed each time the language evolves. See [3] for a survey of metamodel and model co-evolution approaches. Note that while this survey covers a wide range of approaches, it is based on a literature survey and does not cover those co-evolution approaches applied in currently used tools. Moreover, as many of the surveyed approaches are ongoing work, Hebig [3] concludes that there is little data for determining their applicability in industrial contexts.

In [3], one class of approaches suggests the use of transformation languages to co-evolve models each time the metamodel changes. The transformations are made for each case and can be partly automatically produced. A second class of approaches is based on identifying predefined co-evolution strategies or allowing users to specify them. A third approach is searching based on model data, not metamodel, to co-evolve the model to the new metamodel. The final approach identified was labeled as identifying complex metamodel changes. While these approaches cover co-evolution, they focus only on changes in metamodels — although Hebig [3] recognizes that evolution also requires the co-evolution of other artifacts such as transformations or constraints. We aim to inspect all aspects of modeling language change with our evaluation framework.

Co-evolution of transformations and generators is seen as a less popular research subject [11]. The likely reasons are that it is considered as a normal language engineering task and does not have such clear implications for the work of modelers. Also, the wide variety of changes that are possible makes it less automatable, although tools could provide some support.

The evolution of concrete syntax seems also to be seen as a language evolution issue: how the mapping between abstract and concrete syntax is maintained. While it is recognized in evaluation frameworks like [11], its influence on existing models seems to be strongly dependent on the particular tooling used.

B. Research on Language and Modeling Tool Co-Evolution

There is relatively little research on how modeling tool support co-evolves alongside the language supported [16].

Publications comparing language workbenches tend to focus on initial language creation phases (e.g. [7][8][9]) and do not address language evolution, nor the required co-evolution of modeling tool support and existing models. This is somewhat surprising: in a study [17] on practitioners' modeling challenges over 60% named evolution of language a challenge in tools. This need for co-evolution also exists in fixed language modeling tools, when either the language changes or the metamodel of the language is refactored significantly. For example, after moving from SysML 1.6 to SysML 2.0 one of the key concepts, 'Block', does not exist anymore and the language is defined differently from in the past [18].

Another recent study [19], focusing directly on DSM and DSL tools, indicated that a tool's ability to update models automatically when the metamodel changes is considered the second most wanted semantic editor feature — the most important being highlighting model elements and associated error messages. Both these features are addressed in this paper.

Studies directly evaluating tools take a wider view of languages than just metamodels, as at least the editor functionality is inevitably visible, and support for co-evolution quickly becomes visible even with just incremental language definition. In studies evaluating the capabilities of Eclipse-based editors [20][16], concrete syntax is also recognized as a part of the language definition. GMF-based tools are found to lack co-evolution support in many ways [20], and Sirius-based editors break or are incomplete in several co-evolution situations [16] (see Table I in Section III below). Both studies are performed and reported in a methodologically rigorous way, allowing others to repeat and validate them. These studies are restricted in the sense that they do not report changes that deal with constraints related to the language definitions. Evaluations [15][16][20] also vary in their classification of change impact:

1. non-breaking: the editor can open ([16]) or models conform to the metamodel ([15]);
2. complete: all metamodel elements have graphical counterparts in the editor ([16]);
3. valid: the editor exposes correct behavior, e.g. one can create a model conforming to the new metamodel ([16]);
4. resolvable: an automatic procedure can restore validity and completeness after a breaking change ([15][16]); '3 sound' in [20] is similar, but others there do not map well.

Less research seems to have been done on evaluating commercial tools applied on an industrial scale. What is industry-scale may of course vary, but we expect models to be large (>100,000 elements), have many language users (several, dozens or hundreds rather than one or a few), and languages to evolve and be used over a long period of time (over a decade).

Reports on industrial use provide another source for inspecting co-evolution — often related to a specific tool. At Philips, language engineers updated instances manually each time the grammar in Xtext changed [21]. This was recognized as a limitation, but was considered feasible for their case as the number and size of instances (models) was small. Since manual processes become tedious, error-prone and costly with larger models, automated solutions are considered mandatory. Another Xtext case, ([22] p. 263) implemented a generator to automate

transformations that could run over many models in a batch. The actual mappings between two metamodels were made manually. A transformation-based approach was also applied with Microsoft DSL tools, for which Avande presented a mapping language as a basis for generating model converters [23].

A report [24] by Siemens indicated that migration scripts were needed to keep existing models working with MPS. Applying them was challenging because users also had their own branches of models. When the language and generators changed, it became hard to maintain tool support, so finally they hosted custom RCP instances of MPS, one per language version, matching each model release branch.

III. EXPLORATORY EVALUATION

Before proposing a new framework, one should evaluate existing frameworks. Di Ruscio et al. [20] applied a set of criteria to language and tool co-evolution in GMF, and Pierantonio et al. [16] used this existing set to evaluate Sirius. Parts of the framework there are also used in other research mentioned earlier. Use of a common benchmark in this way is a good example of increasing maturity — even if we also want to continue and improve the benchmark, e.g. to cover co-evolution of models as well as tools.

As an exploratory step, we thus took the 11 criteria tested on both GMF and Sirius in the evaluations above, and applied them to evaluate MetaEdit+ (5.5 Build 47). The results are shown in Table I. The coloring is green and o for full success, red and x otherwise, and where Sirius and MetaEdit+ property tests were performed for both simple attributes and more complex references, orange and xo for full success only on attributes.

The 11 criteria of the existing framework were relatively easy to interpret in the context of a different tool. MetaEdit+ modeling tools handled all the changes well, with no errors or omissions in tool behavior. Although the successful co-evolution results were encouraging, they reveal the need for more in-depth evaluation to identify cases, in MetaEdit+ and other tools, where co-evolution support needs more work. Three possible areas of extension can be seen:

- **Location of Change:** The existing framework only tested changes to the abstract syntax of the language, and this should be extended to cover changes made to other

parts of the language, such as its concrete syntax, and rules or constraints.

- **Nature of Change:** The existing framework only considered certain kinds of change operations, so other kinds of operations should be examined to see if they might raise their own questions of co-evolution.
- **Location Impacted by Change:** The existing framework only tested the impact of changes on the tooling, and this should be extended to test the impact on other parts of the language definition (partly covered in [16]), the generators and transformations, and existing models.

IV. FRAMEWORK FOR EVALUATING CO-EVOLUTION SUPPORT

We separate the co-evolution of languages, models and tools into four aspects. The first aspect is the location of the change, i.e. the part of the modeling language being changed: its abstract syntax, constraints, or concrete syntax. These commonly accepted parts of languages are recognized as evolving by others too (e.g. [11]).

The second aspect is the nature of the change: adding, renaming, removing or changing parts of the language definition. These first two aspects thus concern the change that is made; the remaining two aspects cover the possible (adverse) impact of each change.

The third aspect is the location impacted by the change, i.e. which artifacts are adversely affected by the change: other parts of the language definition, the tool support for modeling, generators, or existing models. As not all changes can be automated without adverse effects, we can also look at the capabilities offered by the tool to support the language designer and/or user through the evolution scenarios.

The fourth aspect is the resulting severity of impact on artifacts, ranging from not opening at all to fully co-evolving. We focus particularly on the user's ability to interact with artifacts via the tools. While tooling too is considered an artifact in its own right, a tool may work properly but be unable to open a certain model that has become adversely impacted by language evolution; in that case we consider the problem to be in the model artifact rather than the tool as an artifact itself.

A. Location of Change in Language Definition

Abstract syntax is typically defined via a metamodel. The metamodel may also express the rules and constraints, or they may be expressed in additional constraint or transformation languages. Using language definitions by OMG as examples, a metamodel in MOF specifies the concepts of the language and constraints are defined with OCL. In our evaluation framework, we separate these parts accordingly.

Constraints and rules may be strictly enforced or then shown as warnings when violated, e.g. with a red icon in a diagram symbol, or a warning in an error list pane of the editor [25]. If the rule is made as part of the concrete syntax, we will consider it there rather than as a constraint. Similarly if the rule is written as a generator to produce an error list, we will consider it as part of the generator. The deciding factor is thus where a particular tool or language engineer chooses to implement it, rather than whether it is semantically like a constraint.

TABLE I. SUMMARY OF METAMODEL CHANGE IMPACT ON TOOLS

	GMF [20]	Sirius [16]	MetaEdit+
1 add concrete class	x	x	o
2 add abstract class	x	o	o
3 insert superclass	o	x	o
4 delete class	x	x	o
5 rename class	x	x	o
6 add property	x	xo	o
7 delete property	x	x	o
8 rename property	x	x	o
9 move property	x	x	o
10 pull up property	x	o	o
11 change property type	x	xo	o

Concrete syntax defines the notation, making models visible for humans and accessible via the user interface of the tool. Depending on the representation style, a model can be a diagram, map, matrix, text, tree, etc. or any of their combinations, and each kind of element there may have its own definition.

It would also be possible to consider language changes that affect the semantics rather than other parts of the language definition. How semantics is defined varies based on the nature of language. For example, if used for producing code, semantics is typically defined via a mapping to a programming language (translational semantics via generator), or models are executed at runtime (interpretative semantics). If the language is mainly targeting communication, sketching or documentation, then semantics is typically defined in a prose definition of the language and its elements (e.g. as with modeling languages like ArchiMate and SysML).

In any of these three approaches to semantics, a change to the semantics is unlikely to break other parts of the language definition or models — in the same sense of tool errors and omissions as used for other language changes. We thus do not test changes made to the semantics, but we will examine whether changes made elsewhere can have an impact on the parts of the language definition concerned with semantics, e.g. a generator breaking after a language concept is renamed.

B. Nature of Change: Add, Rename, Remove, Change

Evolution can happen for example by adding, renaming, removing or changing part of the language definition [15]. Looking more closely we can identify:

- Create + **Add** Link (e.g. new kind of object in language)
- Change simple content (e.g. number in constraint)
- **Rename**
- **Remove** Link
- **Change** Link (e.g. A->B becomes A->C)
- Delete (full deletion)
- Change in hierarchy (e.g. pull up property)
- Change of metatype (e.g. relationship becomes object)
- Change simple type (e.g. string becomes int)

A Link is a reference to another first-class element in the metamodel, e.g. that Use Case diagrams can include Actor objects. The reference can either be direct or by name, with the latter generally being brittle with respect to rename operations, but offering indirection and modularization needed in some cases.

Some of the changes listed are so simple that they should cause no problems in tools or models (e.g. creating a new object type). Others are known to be hard, but familiar from many other branches of software engineering (e.g. a string becomes an int). We will focus on the four changes in bold, which in our experience are the key changes encountered in language evolution [26].

We decided not to include changes that are more in the solution domain (refactorings of the metamodel, particularly its inheritance hierarchy) rather than the problem domain (what is desired in the language). As seen in the exploratory evaluation, the definition and details of metamodel refactorings are more

dependent on the language workbench and metamodel, and harder to interpret consistently across different tools: not all tools even allow inheritance within a metamodel. Our experience is that refactorings of this kind tend to occur more often at an early stage, before there are enough models to make co-evolution a question. The same results in the language can also normally be achieved by other means, e.g. rather than pulling up a property to a superclass, it can be added to sibling classes: less ideal, but not as serious an issue as not being able to add, rename, remove or change parts of the language itself.

Before moving on from the aspects about the change itself to the aspects covering the possible (adverse) impact of each change, we should consider our practical philosophy for co-evolution. Where a language change reduces the set of legal models, it is rarely a good idea to adopt a strict formalist approach: e.g. deleting parts of models that no longer correspond to the language definition. The deleted parts would contain information and earlier choices that the modeler will often want to see as part of model co-evolution. Since the models have been legal with respect to the earlier language definition, and valid for generation, a better approach is deprecation: allow the old style, but show warnings and guidance on the new style. This can be accompanied by information on how the deprecation will proceed, e.g. initially allowing both old and new, then not allowing creation of further instances of the old style, then showing warnings for the old style, then making the old style fail integrity checks. Particular cases may need more detailed conditions, e.g. only allow generation targeting products that are themselves sunset or in maintenance mode, or change an old property to be read-only or hidden. In most cases the overall aim should be to guide users towards migrating their behavior and existing models to follow the new approach, but there can also be good reasons to allow the old style to remain, e.g. in models that are no longer actively updated, but still in use. When there is a separate reason to update one of those models, it can be updated to the new style first.

C. Location Impacted by Change

While we focus here mainly on co-evolution impact on models and modeling tools, a change in one part of the language definition may have an impact on other parts of the language definition. For example, in a typical language engineering task adding a new kind of object to a diagram type generally leads to giving it a symbol as its concrete syntax, adding some rules for it, and updating generators to produce code from it. Similarly removing it from the diagram type may leave no longer needed rules and parts of the generator. In both kinds of cases, we will not consider it a problem if the editors work without errors.

Co-evolution within the language definition is not as significant as co-evolution with models, since it influences only the work of a few language engineers. Also, the size of specifications is smaller in language definitions than in system or software specifications in models. It is nevertheless an important aspect, as limited tool support for language evolution can hinder its refinement leading to its stagnation.

The most-studied locations impacted by changes in a language definition are the modeling tools and models. As we have discussed these in depth earlier, there is little to add here.

We will just note that by models we refer to the actual model data, not the ability to view or edit it; that will be considered as part of the modeling tool functionality. A hard dividing line may of course be difficult to set.

Finally, the semantics of the language may be adversely impacted by changes elsewhere in the language definition. As mentioned earlier, we will ignore changes that are simply not made yet: e.g. when adding a new language concept, there will generally be no generation for it, but this is not considered as an error. However, if existing generators now break because of the language change, that is a clear adverse impact.

D. Assessing Change Impact

Since our focus is on tools' capabilities, the framework is made primarily to evaluate how a given tool can cope with the changes. Tool evaluations [15][16][20] characterize tool functionality in a variety of ways, as mentioned earlier. Some of the semantics of those categories seem somewhat unclear, and indeed they seem to be applied somewhat differently in different papers. We will try to follow similar ideas and ordering, but give more concrete descriptions distinguishing the capabilities of editor functionality. Rather than limiting the framework to models, we will use the term 'artifact' to cover the various parts of the language definition, or generators, or models — either existing artifacts made earlier, or creation of a new artifact of that type in the context of this language. The scoring is:

1. When creating a new artifact, the editor does not open, or gives tool errors or warnings.
2. Editor opens for creating a new artifact but does not provide the functionality expected.
3. Editor allows creating a new artifact but support for viewing and editing earlier artifacts is incomplete.
4. Editor opens and asks for human intervention to finalize co-evolution of earlier artifacts.
(4½ if existing models behave and generate correctly, and deprecation guidance is provided as needed.)
5. Editor opens with fully co-evolved earlier artifacts.

Our main focus will be on model artifacts and modeling tool behavior, but we will evaluate for impact on other artifacts too, and the overall score will be the lowest of those for the various kinds of artifact.

E. Scenarios of Co-evolution

Table II shows every possible combination of location and nature of change: the scenarios we want to test. While these could be evaluated individually, a coherent sequence of changes gives a more realistic test. We thus order them to form 12 steps

TABLE II. LOCATION OF CHANGE VS. NATURE OF CHANGE

Location of Change ↓	Nature of Change			
	Add	Rename	Remove	Change
Metamodel	1	4	7	10
Constraints	2	5	8	11
Notation	3	6	9	12

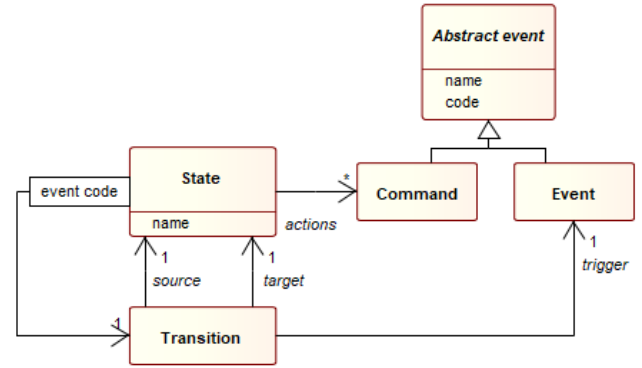


Fig. 1. Metamodel of state machine.

or scenarios, in a sequence similar to what we might see in practice. For example, scenario 1 refers to adding an element to the metamodel, and scenario 2 adds a constraint related to the new element.

For each such scenario we evaluate the impact of the change on other parts of the language definition, on the tool's modeling functionality, on generators, and on existing models. We also evaluate how the tool supports the language developer and language user in the change.

V. AN EXAMPLE LANGUAGE AND MODEL

To make the evaluation concrete and repeatable we use an example from [27]: a state machine for Gothic Security, modeling the secret doors and revolving bookcases of spy films. Its metamodel is shown in Fig. 1 above using a class diagram, with an example model in Fig. 2 below. It should be easy to

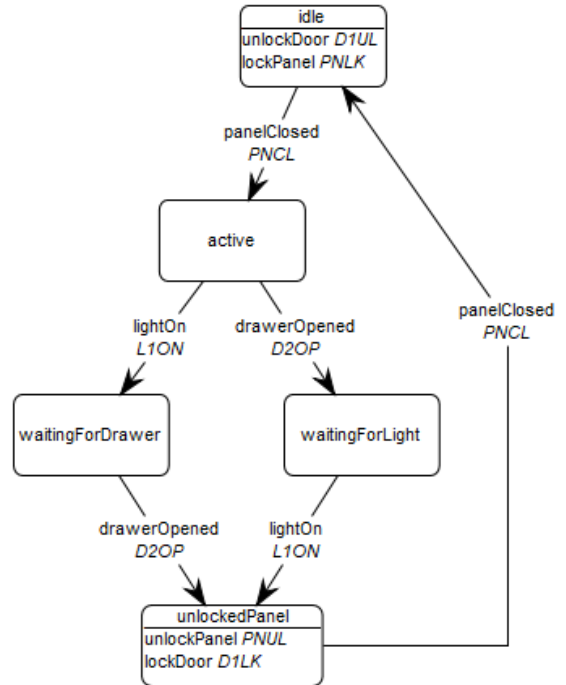


Fig. 2. Example state machine model.

implement in any DSM language workbench. The language is a dialect of state machines: states may have commands, and transitions between states have a triggering event. In [27] both commands and events have a name and a code. There are also constraints, evident only from the generated code, e.g. state name is mandatory and unique within the current state machine.

The model shown in Fig. 2 defines the functionality of a system for Miss Grant for opening a hidden panel [27]. The figure also illustrates the concrete syntax of the language. From a model in this language code can be generated for various targets; when assessing generator co-evolution, we will consider the Java generators.

Following the co-evolution framework presented in Section IV and its Table II, we have 12 different scenarios to inspect whether a given change has adverse impacts on other parts of the language, modeling tools, generators or models. For our example language we choose these concrete scenarios:

1. Add element to metamodel: Add a new Reset element to State machine, with a set of events that trigger it.
2. Add constraint: Only one Reset can be defined in a State machine, and it can connect to only one State there.
3. Add notation: The symbol for Reset is created.
4. Rename element in metamodel: State is renamed to Situation.
5. Rename constraint: In MetaEdit+, constraints do not have names, so no change is needed.
6. Rename notation: The symbol for Situation is renamed.
7. Remove element from metamodel: The Reset element is removed from State machine.
8. Remove constraint: Reset is not allowed to have a relationship to Situation.
9. Remove notation: Reset's symbol is removed.
10. Change metamodel: The Transition relationship's Trigger property is moved to the Source role.
11. Change constraint: Add Start, then update old Reset constraints to point to Start instead, and add Start into the original Transition binding.
12. Change notation: Make the Situation symbol refer to a different library symbol.

Other concrete scenarios would be possible, but these 12 steps are defined so that they can be implemented following each other. In this sense there are 12 sequential versions. All the suggested changes are also evolutionary and not revolutionary: If the language were to change completely, it would be more the case that language engineers would create a new language.

VI. EVALUATION OF METAEDIT+

We show the viability of the framework by applying it to evaluate MetaEdit+ [10][28]. MetaEdit+ is a mature language workbench that supports graphical diagram, matrix and table representations. It enables collaborative work on both language engineering and language use: Multiple people can edit the same language definition and multiple people can use the language at the same time. MetaEdit+ can be used as local installations or remotely in the cloud [29]. MetaEdit+ is commercially successful, used by customers in both industry and academia [30] and is available to download at metacase.com.

Language development and its maintenance can be carried out in MetaEdit+ in three different ways. The primary way, used here, is to use the integrated metamodeling tools in MetaEdit+ Workbench, covering abstract syntax, constraints, concrete syntax and semantics of modeling languages. The second way is graphical metamodeling, where a normal MetaEdit+ model automatically produces and processes the input for the third way, an XML import/export format for metamodels. The graphical way covers the abstract syntax and constraints of the language; the other ways cover all parts.

Detailed results of the evaluation have been made available at <https://github.com/mccjpt/Gothic> as supplementary material, with language definitions and models versioned before and after each co-evolution scenario.

A. Adding New Language Elements: scenarios 1–3

Adding new elements to a language is typically easy from a model co-evolution point of view, as instances of the elements do not yet exist. To add a new metamodel element (scenario #1) in MetaEdit+, the Graph Tool is used to add a new object type 'Reset' with a new property type containing a collection of Events ('Event' already exists in the metamodel). MetaEdit+ provides the editing functionality automatically, along with a simple default notation that the language engineer may change as desired. As a normal language engineer's task, adding a new element to the metamodel may often be followed by related changes to constraints and generators, but these are not necessary for correct tool behavior.

Constraints set well-formedness rules to the language. In MetaEdit+, constraints include 1) bindings that say a relationship type can connect certain types of objects in certain types of roles, possibly via certain types of ports on the object, and 2) constraints on object occurrence, connectivity, ports and property uniqueness.

New constraints (#2) are added in the Graph Tool: an occurrence constraint that allows only one 'Reset' in a graph, and a connectivity constraint that allows a Reset to only be in one Transition. When a constraint is added, its influence on the existing models may need to be checked, as there may be models that do not satisfy the new constraint, e.g. by already having multiple Resets. In MetaEdit+ both models and metamodels are stored in the same repository, allowing language engineers to view and inspect the impact of their changes on models, before committing the changes and making them available for language users. This helps the language engineer experiment, see the results of changes, and think what might be best from a modeler's point of view.

To assist in updating, all models calling for a modeler's decision can be listed or annotated. In MetaEdit+ this could be accomplished by symbol annotation via the Symbol Editor, or by a generator listing model elements that do not meet the constraints. An example of this is shown at the bottom of Fig. 3. In this case, MetaEdit+ would report on models having Resets that do not meet the constraints. This fulfills the tool feature recognized as the most important in [19]: to highlight model elements and associated error messages.

Adding new notation (#3) is straightforward from the co-evolution point of view as models do not yet exist. In MetaEdit+, the symbol for ‘Reset’ is created in the Symbol Editor by drawing it as vector graphics or importing it from an SVG or bitmap file.

As a result of adding these three language elements, editors have full functionality, and all existing models open and update automatically. In the case of a new constraint, modelers are guided to update model elements that violate the constraint.

B. Renaming Language Elements: scenarios 4–6

In language workbenches in general, renaming an element in the metamodel may influence concrete syntax, constraints on the element, and often how semantics is defined. Moreover, it influences existing models. In MetaEdit+ renaming ‘State’ to ‘Situation’ (#4) in the metamodeling tool automatically updates the definitions of related constraints. If there are related generators, they need to be updated with find and replace. If there are several languages using ‘State’ the search can be limited to a given language. Updating generators is not needed if the generator is not bound explicitly to the name of the metamodel element. After renaming an element in the metamodel the models and editor updates automatically.

Renaming a constraint (#5) does not occur in MetaEdit+, as constraints do not have names. If a constraint itself is based on metamodel elements which have been renamed, they were updated automatically earlier in scenario #4.

Renaming symbols (#6) is also tool specific. Typically, symbols in MetaEdit+ are directly related to language elements and do not have names. For more complex cases, a symbol can however also be stored by name in a library, and another symbol can incorporate it from the symbol library by referencing it by name in a template. By renaming a ‘Rectangle’ symbol to ‘BlackRectangle’ in the place where it is referenced, both the rename and reference update are accomplished in one operation. After this update the notation is automatically reflected to models and modeling tools.

To summarize, after renaming scenarios, all tools of MetaEdit+ have full functionality and models are fully updated automatically. In one scenario, generators required simple updates.

C. Removing Language Elements: scenarios 7–9

Removing an element from the metamodel (#7), like ‘Reset’, typically impacts other parts of the language and models. However, since working in the same MetaEdit+ repository “live” with models, before removing anything, language engineers can first consider if it is better just to hide the metamodel element or make it no longer instantiable, rather than delete it and all its instances permanently. This is something that is normally difficult with textual programming languages but which modern tools for language development can provide. This approach allows existing model data to be used for example when generating code — after all, the generator support for them already exists and works. This approach of deprecating rather than hard deletion allows language users to see and update design data created earlier, while guiding them not to use the old

language concept anymore. One bonus here is that if it is later found that removal was not a good idea it is possible to bring the removed parts back — and with good tool support this will also fully restore their instances.

Removing Reset from the metamodel removes it from the language definition and from editor functionality. This operation is done in Graph Tool by removing ‘Reset’ from the used language elements. On the model level, instances are still visible and the language engineer can remove them from models. If the removal involves decisions dependent on the model context, the language engineer can implement model check functionality similarly to that made earlier when adding new constraints. If the removal calls for changes in generators, then the language engineer can implement those similarly to any other generator change.

Removing a constraint or binding (#8), e.g. that ‘Reset’ is allowed to be connected to ‘Situation’, is done in MetaEdit+ by removing it from the list of bindings in Graph Tool. Removing a constraint generally broadens the set of possible models, and so does not require additional actions from the language engineer nor from language users, but removing a binding narrows the set of possible models, so it may be useful to provide deprecation guidance as in scenarios #2 and #7.

Removing an element from the metamodel normally removes its notation automatically too. If only the notation is removed, as with Reset’s symbol in scenario #9, the default notation will be used in its place. If the removed symbol is part of another symbol, like the compartment of commands in ‘Situation’, the language engineer must update the reference (or accept that this part of the symbol will be empty). This is a normal language engineering task rather than an adverse impact.

Removing language elements in scenarios #7-9 calls for normal language engineering tasks. Since deprecation guidance is provided for scenarios #7 and #8, and models, tools and generators continue to work, language users do not necessarily need to take any actions in these cases.

D. Changing Links on Existing Language Elements: #10–12

Changing a link to an existing element in the metamodel, like in #10 moving the ‘Trigger’ Event property from the ‘Transition’ relationship to the ‘Source’ role, is more challenging than a simple removal and addition. In this case the model co-evolution could in theory be automated, as each Transition has exactly one Source role (see Fig. 4 for example code). Deprecation can still be used to good effect: we can allow the Trigger Event property to remain in the Transition, as well as adding it to Source. In that case, it seems most sensible to make the new property (when provided) override the old, and to flag as errors or at least warnings cases where both are provided — at least if they specify different Events.

After this change the ‘Transition’ relationship has still also the ‘Trigger’ information and generators use that data too. Keeping ‘Trigger’ in ‘Transition’ is useful for transition phase so that current Trigger information can be moved to ‘Source’ role. This can be done manually by cutting and pasting the existing Trigger Event from the Transition to the Source role, or by using model transformation with the MetaEdit+ API [28].

Language engineers can also prevent creating new ‘Triggers’ in ‘Transitions’ by making the property type read-only there. They can also give deprecation guidance with an annotation or report as in #2.

Changing an existing constraint calls for changing a link to an existing element. To conduct scenario #11, the language engineer must first add a new object type (‘Start’), in a similar way to scenario #1. Next in the Graph Tool the ‘Start’ is added to the existing binding constraint by including it in the ‘Source’ role alongside ‘Situation’. To finalize the scenario, the existing constraints set in step #2 for ‘Reset’ are updated by changing them to ‘Start’.

Finally, changing a notation link(#12) means choosing another symbol for the notation or its parts. In MetaEdit+ the template subsymbol can be replaced by opening the Symbol Editor for ‘Situation’, opening the template element and changing it to use another subsymbol from the library. (The symbol deliberately uses a template so we can prove the more complex case.)

During the evolution through these changes, editors continue to work without errors or omissions, and old models open automatically. For scenario #10, modelers cannot add Trigger information to Transitions anymore and they see notifications to update the models. If model transformation is used for #10 existing models can also be updated automatically, moving Trigger information to the Source role.

E. Summary

Fig. 3. illustrates the co-evolved language and model after the 12 scenarios. It also includes an extra Situation called

TABLE III. METAEDIT+ CO-EVOLUTION EVALUATION SCORES: METAMODEL, CONSTRAINTS, NOTATION | GENERATOR, TOOL, MODEL

Location of Change ↓	Nature of Change			
	Add	Rename	Remove	Change
Metamodel	5 555 555	4 555 455	4½ 555 554½	4½ 555 554½
Constraints	4½ 555 554½	—	4½ 555 554½	5 555 555
Notation	5 555 555	5 555 555	5 555 555	5 555 555

‘NewState’, to show how the constraint it violates is reported at the bottom of the editor.

Table III summarizes the results of the evaluation. The overall score for each scenario is given first in bold, followed by the score for individual impact locations. (See caption and IV.D on scoring.) The coloring is light green for fully automated (5), lightish green for automation as full as seems desirable (4½), and mid-green for cases in which the only adverse impact is in generator or model co-evolution, where human interaction is needed (4). In none of the cases does the functionality of editors or other tools in MetaEdit+ break or show incorrect or non-working UI elements. In the case of multiple people using the language the result would be the same: they all automatically get the updated language version and same co-evolution success.

In no scenario is there an adverse impact on the metamodel, constraints or notation, nor on the tool functionality, so the co-evolution score for these is always the highest, 5. In most cases the models too update automatically when the language is changed. In no cases are the models damaged, but in three cases the change is such that a fully automatic model update would not be desirable, and in one case not even possible, so deprecation advice is provided for models using the old style, and both new and old style can coexist and generate code correctly. Since existing models and generated code remain valid and deprecation guidance is provided, these cases have a co-evolution score of 4½ in Table III. In scenario 4, the renaming of an element in the metamodel requires a manual find and replace to update the generators, giving a co-evolution score of 4.

Where deprecation with manual update advice was provided, an alternative would be to automate model transformations with the MetaEdit+ API. The API was not used in the co-evolution cases described here, but if used it would change the score to 5 in scenarios 7, 8 and 10, where automation without additional modeler input may be acceptable (see VIII.A for an example).

VII. APPLICABILITY OF THE TOOL EVALUATION FRAMEWORK

Applying the evaluation framework showed that it is viable. The scenario for renaming constraints was not relevant in MetaEdit+, but this may be a good indication of the benefits of deriving the scenarios from first principles, rather than tailoring them to MetaEdit+: other tools may well name constraints.

Completing the evaluation framework’s scenarios was straightforward in MetaEdit+, both for implementing the language changes and assessing their impact. The evaluation was not time-consuming, taking 32 minutes to implement the 12 steps. Each step was completed before the next, including any

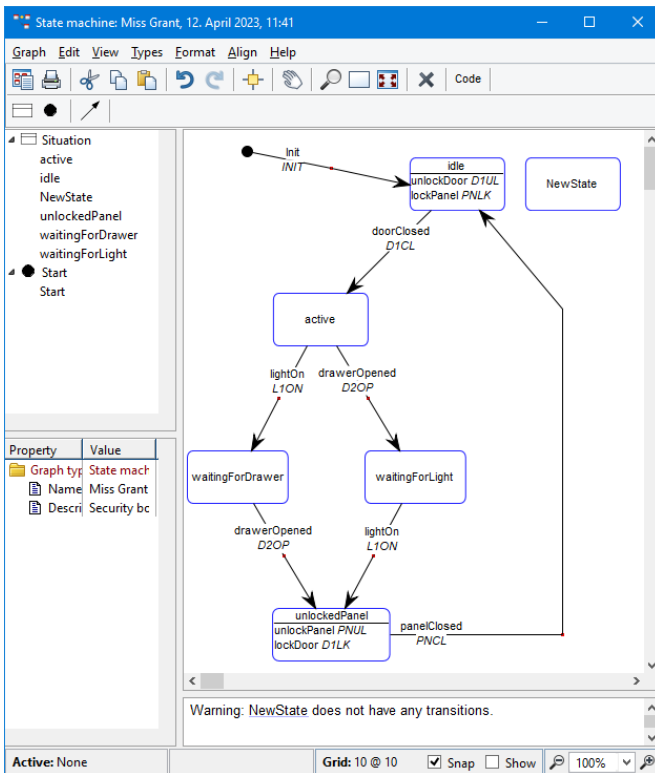


Fig. 3. Model after 12 co-evolution scenarios in MetaEdit+.

necessary manual updates to models or generators, and writing a version comment. This time thus includes the language engineer's work to add co-evolution guidance and update any generators impacted, and the language user's work to update the models according to the co-evolution guidance.

Clearly, this evaluation framework is significantly more stringent than earlier frameworks, covering more aspects and over the whole extent of the language, tools, generators and models. This allowed us to find cases with room for improvement in MetaEdit+, as opposed to its full marks on earlier frameworks in Table I.

VIII. INDUSTRIAL EXPERIENCES OF EVOLUTION

A. When to Use Automation vs. Deprecation

As mentioned earlier, our experience suggests that where language changes such as removals reduce the set of legal models, organizations prefer an approach based on deprecation rather than strictly formalist attempts to automatically or semi-automatically update models wholesale. Conversely, where a change like adding a new language element can indeed be applied automatically with no danger, a completely automatic and largely invisible approach is preferred.

This follows experience with other languages such as those for programming, and also for constructs with language-like usage patterns of few definers and many users, such as libraries or APIs. It is also analogous to a familiar process in natural language: the linguistic concept of a grammatical construct ceasing to be productive, i.e. no longer being used to form new material. When a construct is no longer productive, existing material does not disappear but remains without changing. For instance, the old plural ending *-en* in English is no longer used to form new plurals, but old forms like *children* and *oxen* are still in use. MetaEdit+ provides strong support for the case where a model includes constructs that are no longer present in its (productive) metamodel, allowing this natural process in modeling too.

Making co-evolution fully automatic in cases where it is always known to work, based on the location and nature of the change, is clearly a positive factor in maintaining sustainability with language evolution. For the remaining cases, deprecation seems to be the strategy favored in industrial use — even where automation facilities are provided. As use is rare, it seems best to offer those facilities in a familiar programming language, rather than require learning a new more domain-specific language. The MetaEdit+ API [28] thus includes co-evolution operations that can be called from virtually any language, e.g. C# in Fig. 4, which raises Scenario 10's score from 4½ to 5.

```

METype graphType = new METype() { name = "State machine" };
METype transitionType = new METype() { name = "Transition" };
METype sourceRoleType = new METype() { name = "Source" };

MetaEditAPIPortTypeClient api = new MetaEditAPIPortTypeClient();

foreach (MEOp graph in api.allGoodInstances(graphType))
{
    foreach (MEOp transition in api.contentsMatchingType(graph, transitionType, false))
    {
        MEOp[] sources = api.rolesForRel(graph, transition, sourceRoleType);
        MEAny trigger = api.valueForLocalName(transition, "Trigger");
        api.setValueForLocalName(sources[0], "Trigger", trigger);
    }
}

```

Fig. 4. MetaEdit+ API code to automate Scenario 10 model co-evolution

B. Real-time Automatic Evolution

In Hebig's classification [3], MetaEdit+ co-evolution identifies changes online, is UI-preserving, uses predefined resolution strategies, and is automatic. Interestingly, no other tool in the analysis has the same properties.

With each individual operation on the language definition, the models currently loaded update automatically to correspond to the structure of the new language version. Models that are not currently loaded in memory are updated automatically during loading. For most changes, the update is left to be performed lazily in future too, only being saved at the point where the model element has to be saved anyway, e.g. because it has been changed in normal modeling activities.

For comparison, the evolution of one major version of one GMF language, 214 changes were identified [31]. Of those 214, 197 could be accomplished using the co-evolution mechanisms offered by COPE [31]. Of the 197, 196 would be accomplished automatically and invisibly by built-in evolution of MetaEdit+ and 1 would require using the MetaEdit+ API to update models.

The MetaEdit+ co-evolution mechanisms are robust with respect to skipping intermediate language versions and updating straight to the most recent version. No problems have been encountered, even with languages that have evolved for over 25 years with hundreds of users and gigabytes of models.

C. Language Workbench & Metametamodel Version Updates

In addition to language changes, language workbenches and their metamodels may also evolve, with similar potential impact on language definitions, tooling, generators and models made on that platform. As mentioned above, the GMF language had 214 changes from version 1.0 to 2.0 [31]; a hand-written migrator was provided for language definitions, but not for models made from them nor for the significant amounts of custom code commonly added to build a GMF-based editor.

MetaEdit+ tool version updates can upgrade any languages, generators and models since the first release of version 2.0 in 1995, with fully automatic upgrades since version 3.0 in 1999 — covering many tool releases and significant updates of its GOPRR metamodel [28]. In industrial use, not providing rock-solid, near-zero effort, low risk upgrades is likely to lead to organizations staying on an earlier tool version, with ensuing dissatisfaction and an increasing possibility of a good language being abandoned and benefits lost—a major hit to sustainability.

IX. CONCLUSIONS

We presented a framework to evaluate tool support for co-evolution of languages and models made with them. The framework builds on and combines previous work, and makes it more stringent. It covers changes in language constraints and concrete syntax as well as in the abstract syntax, and the impact on all parts of the language and generators, as well as modeling tools and models. Its scoring offers a more nuanced scale, and hopefully one which is easier to apply consistently across different parts of the modeling ecosystem and different tools.

The evaluation of MetaEdit+ shows that editors do not break and existing models continue to work, largely avoiding the need to create transformations to co-evolve models. The majority of

updates are achieved through automatic, built-in co-evolution of models and modeling tools, with the rest becoming simpler by following the established practice of deprecation rather than deletion. Although providing deprecation guidance requires some work, the overall time of 32 minutes for 12 scenarios indicates that the work is not a significant burden. Some improvements could be made in the existing model co-evolution, e.g. offering better updates for generators after renaming a metamodel element.

The main threats to validity of a new evaluation framework are whether it can be applied to give consistent results for a given tool, and whether it offers a useful comparison across different tools. The trial evaluation of MetaEdit+ was performed by Tolvanen and checked by Kelly: both experienced and thus reliable test subjects, and using the normal tool functions available to all users. Testing a previous framework made for other tools and building on it improved cross-tool applicability. Just as we built on an earlier framework, so our framework too could be improved and verified by wider application. We thus invite others to repeat the evaluation described here and to apply it to evaluate other tools. More extensive cases are also welcome, along with others' experience of industrial-scale use.

Making language evolution simple, low-effort and low-risk can be seen to better ensure that languages evolve to sustainably continue to meet development needs. In industrial use of MetaEdit+ since its first version in 1995, this approach has compared favorably to approaches and tools that require metamodelers and/or modelers to implement and apply transformations each time the language changes.

REFERENCES

- [1] J. Sprinkle, M. Mernik, J.-P. Tolvanen and D. Spinellis, "What kinds of nails need a domain-specific hammer?", *IEEE Software*, July/Aug. 2009.
- [2] S. Kelly and R. Pohjonen, "Worst practices for Domain-Specific Modeling", *IEEE Software*, vol. 26, no. 4, 2009.
- [3] R. Hebig, D. Khelladi and R. Bendraou, "Approaches to co-evolution of metamodels and models: a survey", *IEEE Transactions on Software Engineering*, vol. 43, no. 5, May 1, 2017.
- [4] H. S. Borum and C. Seidl, "Survey of established practices in the life cycle of Domain-Specific Languages", *Proc. 25th Int. Conf. Model Driven Engineering Languages and Systems (MODELS '22)*, ACM, 2022.
- [5] P. Lago, S. A. Koçak, I. Crnkovic and B. Penzenstadler, "Framing sustainability as a property of software quality", *Commun. ACM*, vol. 58, no. 10, Oct. 2015.
- [6] DSMForum, <http://dsmforum.org/cases.html> (accessed April 2023).
- [7] S. Erdweg, T. van der Storm, M. Voelter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth and J. van der Woning, "The State of the Art in Language Workbenches", *Software Language Engineering (SLE 2013)*, LNCS, vol. 8225. Springer, Cham. 2013.
- [8] A. El Kouhen, C. Dumoulin, S. Gérard and P. Boulet, "Evaluation of Modelling Tools Adaptation", CNRS HAL hal-00706701, 2012. <http://tinyurl.com/gerard12>
- [9] J.-P. Tolvanen and S. Kelly, "Effort used to create Domain-Specific Modeling languages", *ACM/IEEE 21st Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS 2018)*, ACM, 2018. <https://doi.org/10.1145/3239372.3239410>
- [10] S. Kelly, K. Lyytinen and M. Rossi, "MetaEdit+: A fully configurable multi-user and multi-tool CASE and CAME environment", *Conf. on Advanced Information Systems Engineering (CAiSE 1996)*, 1996.
- [11] B. Meyers and H. Vangheluwe, "A framework for evolution of modelling languages", *Science of Computer Programming*, vol. 76, no. 12, 2011.
- [12] G. Wachsmuth, "Metamodel Adaptation and Model Co-adaptation", *European Conf. on Object-Oriented Programming*, 2007.
- [13] Y. Xiong, D. Liu, D. Hu, H. Zhao, M. Takeichi, and H. Mei, "Towards automatic model synchronization from model transformations", *Proc. 22nd IEEE/ACM Int. Conf. Automated Software Engineering (ASE '07)*, ACM, 2007.
- [14] B. Gruschko, D. Kolovos and R. Paige, "Towards synchronizing models with evolving metamodels", *Proc. Int. Workshop on Model-Driven Software Evolution*, IEEE, 2007.
- [15] A. Cicchetti, D. Di Ruscio, R. Eramo and A. Pierantonio, "Automating co-evolution in model-driven engineering", *Enterprise Distributed Object Computing Conference, 2008. EDOC '08*, pp. 222–231, IEEE, 2008.
- [16] A. Pierantonio, J. Di Rocco, D. Di Ruscio and H. Narayanankutty, "Resilience in Sirius editors: Understanding the impact of metamodel changes", *ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS 2018)*, 2018.
- [17] M. Ozkaya and F. Erata, "Understanding pPractitioners' challenges on software modeling: A survey", *J. Computer Languages*, vol. 58, 2020.
- [18] OMG, "Systems Modeling Language, SysML v1 to SysML v2 Transformation", Release 2023-02, 2023.
- [19] M. Ozkaya and D. Akdur, "What do practitioners expect from the meta-modeling tools? A survey", *J. Computer Languages*, vol. 63, 2021.
- [20] D. Di Ruscio, R. Lämmel and A. Pierantonio, "Automated Co-evolution of GMF editor models", *Int. Conf. on Software Language Engineering*, pp. 143–162, Springer, 2010.
- [21] M. Schuts, M. Alonso and J. Hooman, "Industrial experiences with the evolution of a DSL", *Proc. 18th ACM SIGPLAN Int. Workshop on Domain-Specific Modeling (DSM 2021)*, ACM, 2021.
- [22] B. Akesson, J. Hooman, J. Sleuters and A. Yankov, "Reducing design time and promoting evolvability using Domain-Specific Languages in an industrial context", *Model Management and Analytics for Large Scale Systems*, Academic Press, 2020.
- [23] G. de Geest, A. Savelkoul and A. Alikoski, "Building a framework to support Domain Specific Language evolution using Microsoft DSL Tools", *Proc. 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07)*, J. Sprinkle, J. Gray, M. Rossi and J.-P. Tolvanen, Eds., Computer Science and Information System Reports, Technical Reports, TR-38, University of Jyväskylä, Finland, 2007.
- [24] D. Ratiu, H. Nehls, A. Joanni and S. Rothbauer, "Use MPS to unleash the creativity of domain experts: Language engineering is a key enabler for bringing innovation in industry". In: *Domain-Specific Languages in Practice*, A. Bucchiarone, A. Cicchetti, F. Ciczozzi and A. Pierantonio, Eds. Springer, 2021.
- [25] S. Kelly and J.-P. Tolvanen, "Automated Annotations in Domain-Specific Models: Analysis of 23 Cases", *1st Int. Workshop on Foundations and Practice of Visual Modeling*, 2021.
- [26] S. Kelly, M. Rossi and J.-P. Tolvanen, "What is needed in a MetaCASE environment", *Enterprise Modelling and Information Systems Architectures*, vol. 1, no. 1, 2005.
- [27] M. Fowler, *Domain-Specific Languages*, Addison-Wesley, 2010.
- [28] MetaCase, *MetaEdit+ 5.5 User's Guides*, <https://metacase.com/support/55/manuals/> (accessed August 2023)
- [29] S. Kelly and J.-P. Tolvanen, "Collaborative modelling and metamodeling with MetaEdit+", *HoWCoM 2021*, MoDELS 2021 Companion, Fukuoka, Japan, 2021. doi:10.1109/MODELS-C53483.2021.00012
- [30] J.-P. Tolvanen and S. Kelly, "Model-Driven Development Challenges and Solutions - Experiences with Domain-Specific Modelling in Industry", *Proc. 4th Int. Conf. Model-Driven Engineering and Software Development (MoDELS 2016)*, 2016.
- [31] M. Herrmannsdorfer, S. Benz and E. Juergens, "COPE - Automating coupled evolution of metamodels and models," in *ECOOP 2009 -- Object-Oriented Programming*, pp. 52–76, Springer, 2009.
- [32] D. E. Khelladi, R. Hebig, R. Bendraou, J. Robin and M.-P. Gervais, "Detecting complex changes during metamodel evolution," *Advanced Information Systems Engineering (CAiSE 2015)*, Springer, 2015.